Adapted from Graham's Lecture slides.

第六章： 递归函数

基本概念，序列上的递归函数，
相互递归

北京大学
PEKING UNIVERSITY

函数的定义和作用

As we have seen, many functions can naturally be defined in terms of other functions.

```
fac :: Int → Int
fac n = product [1..n]
```

fac maps any integer n to the product of the integers between 1 and n.

Expressions are evaluated by a stepwise process of applying functions to their arguments.

For example:

```
  fac 4
=
  product [1..4]
=
  product [1,2,3,4]
=
  1*2*3*4
=
  24
```

递归函数

In Haskell, functions can also be defined in terms of themselves. Such functions are called recursive.

```
fac 0 = 1
fac n = n * fac (n-1)
```

fac maps 0 to 1, and any other integer to the product of itself and the factorial of its predecessor.

For example:

```
  fac 3
=
  3 * fac 2
=
  3 * (2 * fac 1)
=
  3 * (2 * (1 * fac 0))
=
  3 * (2 * (1 * 1))
=
  3 * (2 * 1)
=
  3 * 2
=
  6
```

Note:

- fac 0 = 1 is appropriate because 1 is the identity for multiplication: 1*x = x = x*1.

- The recursive definition diverges on integers < 0 because the base case is never reached:

```
> fac (-1)

*** Exception: stack overflow
```

递归函数的作用

- Some functions, such as factorial, are simpler to define in terms of other functions.

- As we shall see, however, many functions can naturally be defined in terms of themselves.

- Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of induction.

序列上的递归函数

Recursion is not restricted to numbers, but can also be used to define functions on lists.

```
product :: Num a ⇒ [a] → a
product []       = 1
product (n:ns) = n * product ns
```

product maps the empty list to 1, and any non-empty list to its head multiplied by the product of its tail.

For example:

```
  product [2,3,4]
=
  2 * product [3,4]
=
  2 * (3 * product [4])
=
  2 * (3 * (4 * product []))
=
  2 * (3 * (4 * 1))
=
  24
```

Using the same pattern of recursion as in product we can define the length function on lists.

```
length :: [a] → Int
length []     = 0
length (_:xs) = 1 + length xs
```

length maps the empty list to 0, and any non-empty list to the successor of the length of its tail.

For example:

```
  length [1,2,3]
=
  1 + length [2,3]
=
  1 + (1 + length [3])
=
  1 + (1 + (1 + length []))
=
  1 + (1 + (1 + 0))
=
  3
```

Using a similar pattern of recursion we can define the reverse function on lists.

```
reverse :: [a] → [a]
reverse []      = []
reverse (x:xs) = reverse xs ++ [x]
```

reverse maps the empty list to the empty list, and any non-empty list to the reverse of its tail appended to its head.

For example:

```
  reverse [1,2,3]
=
  reverse [2,3] ++ [1]
=
  (reverse [3] ++ [2]) ++ [1]
=
  ((reverse [] ++ [3]) ++ [2]) ++ [1]
=
  (([] ++ [3]) ++ [2]) ++ [1]
=
  [3,2,1]
```

课堂练习

- 给出下面程序中的insert的类型和定义，完成“插入排序”算法的定义。

```
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

多参数递归

Functions with more than one argument can also be defined using recursion. For example:

- Zipping the elements of two lists:

```
zip :: [a] → [b] → [(a,b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Remove the first n elements from a list:

```
drop :: Int → [a] → [a]
drop 0 xs     = xs
drop _ []     = []
drop n (_:xs) = drop (n-1) xs
```

- Appending two lists:

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

多重递归 (Multiple Recursion)

Functions can also be defined using multiple recursion, in which a function is applied more than once in its own definition.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-2) + fib (n-1)
```

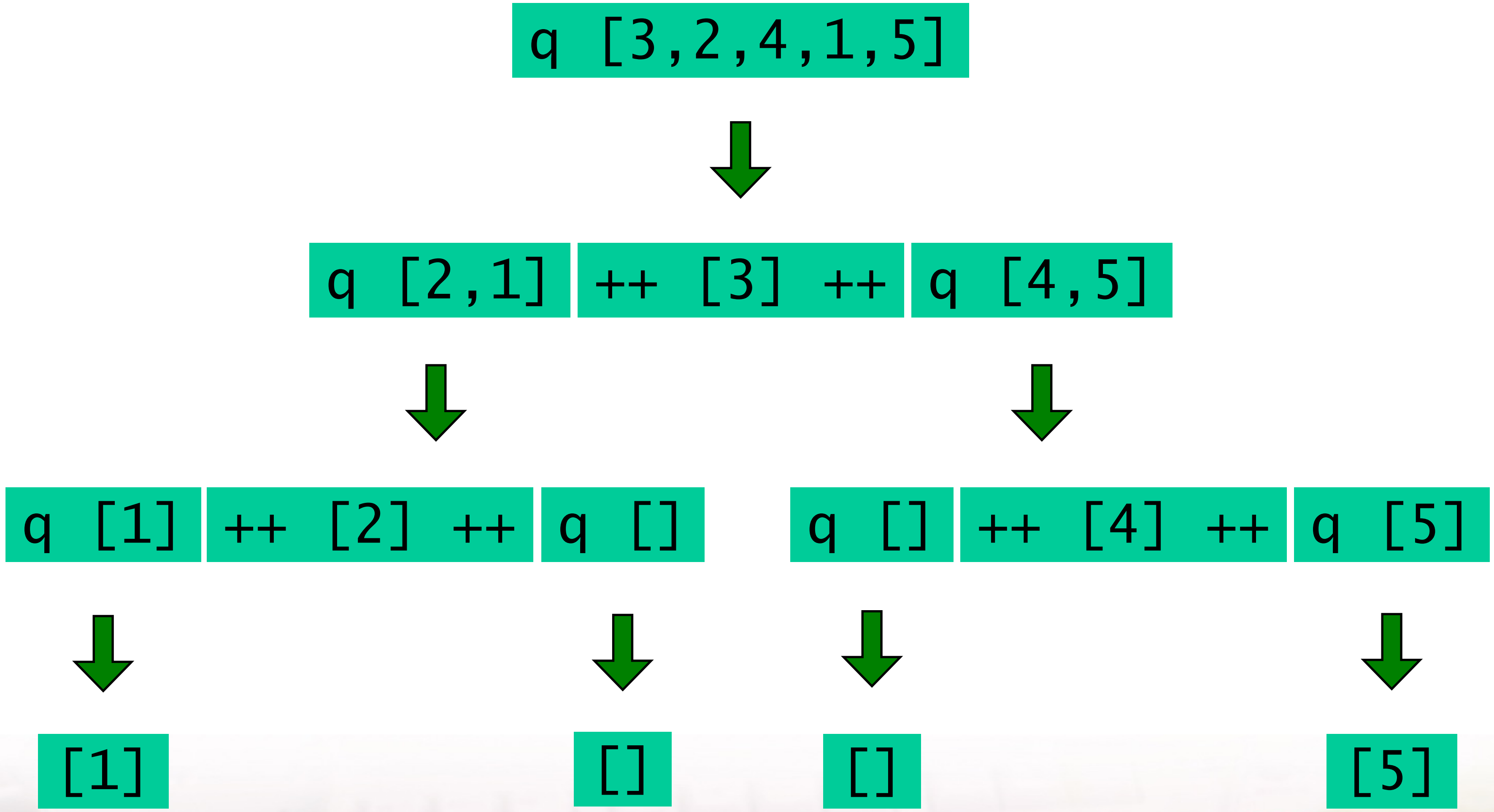
快速排序:

```
qsort :: Ord a ⇒ [a] → [a]
qsort []       = []
qsort (x:xs) =
   qsort smaller ++ [x] ++ qsort larger
   where
      smaller = [a | a ← xs, a ≤ x]
      larger  = [b | b ← xs, b > x]
```

Note:

- This is probably the simplest implementation of quicksort in any programming language!

For example (abbreviating qsort as q):



相互递归（Mutual Recursion)

Functions can also be defined using mutual recursion, in which two or more functions are all defined recursively in terms of each other.

```
even :: Int -> Bool
even 0 = True
even n = odd (n-1)

odd :: Int -> Bool
odd 0 = False
odd n = even (n-1)
```

作业

6-1 Without looking at the standard prelude, define the following library functions using recursion:

- Decide if all logical values in a list are true:

```
and :: [Bool] → Bool
```

- Concatenate a list of lists:

```
concat :: [[a]] → [a]
```

- Produce a list with n identical elements:

```
replicate :: Int → a → [a]
```

- Select the nth element of a list (starting from 0):

```
(!!) :: [a] → Int → a
```

- Decide if a value is an element of a list:

```
elem :: Eq a ⇒ a → [a] → Bool
```

6-2 Define a recursive function

```
merge :: Ord a ⇒ [a] → [a] → [a]
```

that merges two sorted lists of values to give a single sorted list. For example:

```
> merge [2,5,6] [1,3,4]

[1,2,3,4,5,6]
```

6-3 Define a recursive function

```
msort :: Ord a ⇒ [a] → [a]
```

that implements merge sort, which can be specified by the following two rules:

- Lists of length ≤ 1 are already sorted;
- Other lists can be sorted by sorting the two halves and merging the resulting lists.